Frama-C Plug-in Developer Training

Virgile Prevosto and Julien Signoles

firstname.lastname@cea.fr

2012-05-xy

Overview

Part 1: First steps

Part 2: Architecture overview

Part 3: Integration into *Frama-C*

Part 4: Writing analyses

- Gentle *OCaml* reminder
  - Basic features
  - Module system
  - Object-oriented features
- My first *Frama-C* script
  - Browsing *Frama-C*'s API
  - Script-driven analysis
  - Detection of `const` violation

OCaml, a functional language

- *OCaml* is a functional language
- functions are first-class values

Basic features

OCaml, a functional language

- *OCaml* is a functional language
- functions are first-class values

```
# let l = [ 1; -4; 5; 3 ];;
```

Basic features

OCaml, a functional language

- *OCaml* is a functional language
- functions are first-class values

```
# let l = [ 1; -4; 5; 3 ];;
val l : int list = [1; -4; 5; 3]
```

OCaml, a functional language

- *OCaml* is a functional language
- functions are first-class values

```
# let l = [ 1; -4; 5; 3 ];;
val l : int list = [1; -4; 5; 3]
# List.map (fun x -> x * 2) l;;
```

Basic features

OCaml, a functional language

- *OCaml* is a functional language
- functions are first-class values

```
# let l = [ 1; -4; 5; 3 ];;
val l : int list = [1; -4; 5; 3]
# List.map (fun x -> x * 2) l;;
- : int list = [2; -8; 10; 6]
```

Basic features

OCaml, a functional language

- *OCaml* is a functional language
- functions are first-class values

```
# let l = [ 1; -4; 5; 3 ];;
val l : int list = [1; -4; 5; 3]
# List.map (fun x -> x * 2) l;;
- : int list = [2; -8; 10; 6]
# List.sort Pervasives.compare l;;
```

Basic features — *OCaml*, a functional language

- *OCaml* is a functional language
- functions are first-class values

```
# let l = [ 1; -4; 5; 3 ];;
val l : int list = [1; -4; 5; 3]
# List.map (fun x -> x * 2) l;;
- : int list = [2; -8; 10; 6]
# List.sort Pervasives.compare l;;
- : int list = [-4; 1; 3; 5]
```

Basic features

OCaml, a functional language

- *OCaml* is a functional language
- functions are first-class values

```
# let l = [ 1; -4; 5; 3 ];;
val l : int list = [1; -4; 5; 3]
# List.map (fun x -> x * 2) l;;
- : int list = [2; -8; 10; 6]
# List.sort Pervasives.compare l;;
- : int list = [-4; 1; 3; 5]
# List.fold_left ( + ) 0 l;;
```

Basic features

OCaml, a functional language

- *OCaml* is a functional language
- functions are first-class values

```
# let l = [ 1; -4; 5; 3 ];;
val l : int list = [1; -4; 5; 3]
# List.map (fun x -> x * 2) l;;
- : int list = [2; -8; 10; 6]
# List.sort Pervasives.compare l;;
- : int list = [-4; 1; 3; 5]
# List.fold_left ( + ) 0 l;;
- : int = 5
```

Basic features

OCaml, an imperative language

Reference and mutable records

- *OCaml* is also an imperative language
- references

```
let x = ref 0
let () = x ← 3
(* let () = x ← "three" *) (* incorrect! *)
let n = !x              (* n is 3 *)
let y = x               (* aliasing *)
let () = x ← 2
let m = !y              (* m is 2 *)
```

- mutable records

```
type t = { mutable int a; bool b }
let x = { a = 0; b = true; }
let () = x.a ← 1
```

Basic features

OCaml, an imperative language

Printers

- printers *à la* printf (even more powerful)

```
let three = "three"
let () = Format.printf "%s is %d" three 3

type t = A | B of int
let print fmt = function
  | A -> Format.fprintf fmt "A"
  | B n -> Format.fprintf fmt "B %d" n
let () =
  List.iter
    (fun x -> Format.printf "%a " print x)
    [ A; B 3; B (-4) ]
```

Basic features

OCaml, an imperative language

Standard imperative datastructures

- Hashtables

```
let h = Hashtbl.create 7
let () =
  List.iter
    (fun (n, s) -> Hashtbl.add h n s)
    [ 1, "one"; 2, "two"; 3, "three" ]
let two = Hashtbl.find h 2
let () =
  Hashtbl.iter
    (fun (n, s) ->
      Format.printf "%d --> %s" n s)
    h
```

- Array, Stack, Queue

Basic features

OCaml, an imperative language

Exception

```
let h = (* hashtbl of the previous slide *)
let four =
  try Hashtbl.find h 4
  with Not_found → "four"

exception Found of string
let mem_value p =
  try
    Hashtbl.iter
      (fun _ s → if p s then raise (Found s))
      h;
    None
  with Found s →
    Some s
```

OCaml, an imperative language

- sharing and backwards links
  - aliasing
  - *Frama-C*'s AST
- complexity
  - random access in an array *vs* in a list
  - search in an hashtbl *vs* in a map or a list
- ease of implementation
  - raising an exception *vs* returning an option type
  - no need to push an environment across call stack

Module system

- small typed functional language by itself
- based on the core language
- namespace
- encapsulation
- generic programming

Module system


```
(* implementation of rationals *)
struct
  type t = int * int
  let pgcd n m = ...
  let make n d =
    let p = pgcd n d in
    n / p, d / p
  let integer n = n, 1
  let add (n1, d1) (n2, d2) =
    make (n1 * d2 + n2 * d1) (d1 * d2)
  ...
end
```

Module system

Module

Names, submodule and access

```
(* modules can be named *)
module Rational =
  ... (* code of the previous slide *)

(* submodules are possible *)
module M1 = struct
  module M2 = struct
    module M3 = struct let x = ... end
  end
end

(* access through the dot notation *)
let r_one: Rational.t = Rational.integer 1
let x = M1.M2.M3.x
```

- *OCaml* infers a module type for each module
- types of structure are signatures

```
(* inferred type for module Rational *)
sig
  type t = int * int
  val pgcd: int → int → int
  val make: int → int → int * int
  val integer: int → int * int
  val add: int * int → int * int → int * int
  ...
end
```

Module system


```
module type Rational = sig
  type t
  val make: int → int → t
  val integer: int → t
  val add: t → t → t
  ...
end
module Rational: Rational
```

- abstract types
- hide implementation details through subtyping
- encapsulation: easy to change the implementation without changing its interface
- unnamed signature

Module system

Opening and inclusion

```
open Rational
let r_one = zero
open Cil_types

module My_list = struct
  include List
  let singleton x = [ x ]
  let tl _ = failwith "should never be used"
end
```

- 'open' provides a direct access to a structure's namespace (or signature)
- usually bad to have to many 'open' at the same time
- 'include' allows to extends/redefine a structure or a signature

Module system

Module

Functor definition

```
module type Ring = sig
  type t
  val zero: t            val one: t
  val add: t → t → t    val opp: t → t
  val mult: t → t → t
end
module Polynomial(R: Ring) = struct
  type ring = R.t      type t = R.t array
  let zero = [| R.zero |]
  let monomial c n =
    let p = Array.create (n + 1) R.zero in
    p.(n) ← c; p
   ...
end
```

Module system


```
module IntegerPolynomial =
  Polynomial
    (struct
      type t = int
      let zero = 0
      let one = 1
      let add = ( + )
      let mult = ( * )
      let opp n = - n
    end)

module RationalPolynomial =
  Polynomial(Rational)
```

Module system


```
module Polynomial(R: Ring) = sig
  type ring = R.t     type t
  val zero: t
  val monomial: R.t → int → t
  ...
end
```

```
module type Polynomial: sig
  type ring      type t
  val zero: t    val monomial: R.t → int → t
end
module Polynomial(R: Ring):
  Polynomial with type ring = R.t
```

A small comparison

	Traditional OO languages	*OCaml*
Encapsulation	Objects	Modules
Late binding	Objects	Objects

Objects in *OCaml*

- used only where one an extensible behavior is explicitly desired.
- modules and functors often more suitable.
- Two usages in *Frama-C*: AST visitor and lablgtk-based GUI

Object-oriented features

How to define a class

```
class my_visitor x y:
Visitor.frama_c_visitor =
let local_var = f x y in
object(self)
  inherit Visitor.frama_c_inplace
  val v1 = Stack.create ()
  val mutable v2 = 0
  method vvrbl vi = ...
      self#internal_method v2
  method private internal_method x = ...
end
```

Object-oriented features

How to define a class

```
class my_visitor x y:
Visitor.frama_c_visitor =
let local_var = f x y in
object(self)
   inherit Visitor.frama_c_inplace
   val v1 = Stack.create ()
   val mutable v2 = 0
   method vvrbl vi = ...
      self#internal_method v2
   method private internal_method x = ...
end
```

Classes can take parameters

Object-oriented features

How to define a class

```
class my_visitor x y:
Visitor.frama_c_visitor =
let local_var = f x y in
object(self)
  inherit Visitor.frama_c_inplace
  val v1 = Stack.create ()
  val mutable v2 = 0
  method vvrbl vi = ...
      self#internal_method v2
  method private internal_method x = ...
end
```

Constrain the interface (class type)

Object-oriented features

How to define a class

```
class my_visitor x y:
Visitor.frama_c_visitor =
let local_var = f x y in
object(self)
   inherit Visitor.frama_c_inplace
   val v1 = Stack.create ()
   val mutable v2 = 0
   method vvrbl vi = ...
       self#internal_method v2
   method private internal_method x = ...
end
```

Inheritance

Object-oriented features

How to define a class

```
class my_visitor x y:
Visitor.frama_c_visitor =
let local_var = f x y in
object(self)
  inherit Visitor.frama_c_inplace
  val v1 = Stack.create ()
  val mutable v2 = 0
  method vvrbl vi = ...
    self#internal_method v2
  method private internal_method x = ...
end
```

Naming current object

Object-oriented features

How to define a class

```
class my_visitor x y:
Visitor.frama_c_visitor =
let local_var = f x y in
object(self)
  inherit Visitor.frama_c_inplace
  val v1 = Stack.create ()
  val mutable v2 = 0
  method vvrbl vi = ...
      self#internal_method v2
  method private internal_method x = ...
end
```

Calling a method

Object-oriented features

How to define a class

```
class my_visitor x y:
Visitor.frama_c_visitor =
let local_var = f x y in
object(self)
   inherit Visitor.frama_c_inplace
   val v1 = Stack.create ()
   val mutable v2 = 0
   method vvrbl vi = ...
      self#internal_method v2
   method private internal_method x = ...
end
```

Normal (public) method

Object-oriented features

How to define a class

```
class my_visitor x y:
Visitor.frama_c_visitor =
let local_var = f x y in
object(self)
  inherit Visitor.frama_c_inplace
  val v1 = Stack.create ()
  val mutable v2 = 0
  method vvrbl vi = ...
      self#internal_method v2
  method private internal_method x = ...
end
```

Private method

Object-oriented features

How to define a class

```
class my_visitor x y:
Visitor.frama_c_visitor =
let local_var = f x y in
object(self)
  inherit Visitor.frama_c_inplace
  val v1 = Stack.create ()
  val mutable v2 = 0
  method vvrbl vi = ...
      self#internal_method v2
  method private internal_method x = ...
end
```

Instance variable

Object-oriented features

How to define a class

```
class my_visitor x y:
Visitor.frama_c_visitor =
let local_var = f x y in
object(self)
  inherit Visitor.frama_c_inplace
  val v1 = Stack.create ()
  val mutable v2 = 0
  method vvrbl vi = ...
      self#internal_method v2
  method private internal_method x = ...
end
```

Mutable instance variable

Object-oriented features

How to define a class

```
class my_visitor x y:
Visitor.frama_c_visitor =
let local_var = f x y in
object(self)
  inherit Visitor.frama_c_inplace
  val v1 = Stack.create ()
  val mutable v2 = 0
  method vvrbl vi = ...
      self#internal_method v2
  method private internal_method x = ...
end
```

Local variables

Object-oriented features

Typing

- Each `class` implicitly defines a `class type`
- Type is mainly the list of public methods with their type
- Explicit definition:

  `class type my_class_type = object ... end`
- Structural subtyping, not directly related to inheritance
  - class A is a subtype of B if it has at least the same methods
  - and the type of method m in A is a subtype of the one in B
  - formalized duck-typing

Object-oriented features

Definition of class members

	Method	Private method	Instance variable	Local binding
Available outside of object	Yes	No	No	No
Available in inherited classes	Yes	Yes	Yes	No
Late binding	Yes	Yes	No	No

Object-oriented features

How to use objects

creation `let obj = new my_visitor x y`

coercion `(obj :> Visitor.frama_c_visitor)`

direct definition `let obj = object ... end`

Browsing *Frama-C*'s API — Reading `.mli` files

Access

- Directly open the desired file in your favorite IDE
- Some interesting files:
  - `cil/src/cil_types.mli`, `cil.mli`
  - `src/kernel/globals.mli`, `kernel_functions.mli`
  - `src/kernel/file.mli`, `visitor.mli`
  - `src/kernel/plugin.mli`, `log.mli`
  - `src/type/datatype.mli`,
  - `src/project/state_builder.mli`
  - `src/kernel/dynamic.mli`, `journal.mli`

Pros and Cons

- ✔ Can be used directly in IDE
- ✘ Requires some knowledge of where functions are
  - Might be mitigated by IDE's OCaml support

Browsing *Frama-C*'s API Generated HTML Documentation

Access

- Not compiled by default: requires `make doc install-doc-code`
- `$FRAMAC_SHARE/doc/code/html/index.html`
- `http://frama-c.com/download/frama-c-Nitrogen-20111001_api.tar.gz`

Pros and Cons

- ✔ Provides various indexes
- ✔ Easier navigation between files
- ✘ No search
- ✘ Generation is costly (but required only once)

Browsing *Frama-C*'s API — OCamlbrowser

Access

- Program included in OCaml distribution (if `tcl/tk` enabled)
- Reads `.cmi` interfaces to provide information
- Requires setting up its search path

Pros and Cons

- ✔ Searchable (including by type)
- ✘ No recursive descent in directories: must give all paths manually

Script-driven analysis

When to use a script?

- replay and/or edit the **journal** of a GUI session
- compose analyses
- access functionalities that can't be done *via* command-line options

Script-driven analysis

Basic usage

- All analyses and options can be accessed programmatically
- Provide a function `run` to set up appropriate environment…
- … and launches the analyses in the desired order
- Register `run` itself as a toplevel analysis
- Sample example

Detection of `const` violation

Scenario

Plug-in's basic elements

- Registering
- Messages
- Options
- Extention Points

Kernel infrastructure

- AST and front-end
- Properties and their statuses
- States and Datatypes

Registering

Registering

Plugin.Register

Messages

Messages

Log

Options

Options

Cmdline; Options

Extention Points

Extension Points

Db.Main.extend – is_computed (avec des ref dans un premier temps)
initialisation

AST and front-end

Representation of a C program

Retrieving information

AST and front-end

From original source to final AST

The freshly parsed AST

Properties and their statuses

Property datatype

Properties and their statuses

Local status

Properties and their statuses

Consolidated status

States and Datatypes

Datatype

States and Datatypes

Registering a new state

API registration

- Datatype
- Dynamic Linking
- Export a Value
- Use a Dynamic API
- Abstract Type
- Journalisation

Project system

- Overview
- State
- Project Operations
- State Selection
- Marshaling

Datatype

Overview

- a *datatype* is a fundamental notion of *Frama-C*
- it provides standard operations for a given type in a single module
- most types used in *Frama-C* have an associated datatype
- many *Frama-C* functors require a datatype as argument
- subsumes the *Frama-C* notion of *type value*, which may be seen as type as first class values

Datatype

Type

- implemented in the low-level module `Type`
- for each monomorphic type `ty`, a (unique) value of type `ty Type.t` dynamically represents the type `ty` as a ML value.
- type values allow to use dynamic typing in *Frama-C* as shown latter.
- type values for basic *OCaml* types are provided in `Datatype`

```
(* extract of datatype.mli *)
val unit: unit Type.t
val int: int Type.t
val string: string Type.t
val formatter: Format.formatter Type.t
...
```

Datatype


```
(* extract of datatype.mli *)
module type S = sig
  type t
  val ty: t Type.t
  val name: string
  val equal: t → t → bool
  val compare: t → t → int
  val hash: t → int
  val copy: t → t
  val pretty: Format.formatter → t → unit
  ... (* other less important functions *)
end
```

Datatype


```
(* extract of datatype.mli *)
module type S_with_collections = sig
  include S
  module Set: Set with type elt = t
  module Map: Map with type key = t
  module Hashtbl: Hashtbl with type key = t
end
```

Datatype

- module `Datatype`: datatypes for basic *OCaml* types

```
(* extract of datatype.mli *)
module Unit: S_with_collections
module Int: S_with_collections
module String: S_with_collections
module Formatter: S
```

Datatype

Existing Datatypes (again)

- module `Cil_datatype`: datatypes for AST types

```
(* extract of cil_datatype.mli *)
module Stmt: sig
  include Datatype.S_with_collections
    with type t = stmt
  ...
end
```

- *Frama-C* data structures usually implement includes at least `Datatype.S`

```
(* extract of property_status.mli *)
include Datatype.S with type t = status
```

Datatype

Datatype

How to create a new one?

```
module Rational = struct
  type rational = { num: int; denom: int }
  include Datatype.Make_with_collections
    (struct
      type t = rational
      let name = "Rational.t"
      let reprs = [ { num = 0; denom = 1 } ]
      include Datatype.Serializable_undefined
      let equal (x:t) y = x = y
      let compare (x:t) y = Pervasives.compare x y
      let hash (x:t) = Hashtbl.hash x
      let copy x = x
      let pretty fmt x =
        Format.fprintf fmt "%d/%d" x.num y.denom
    end)
  ...
end
```

Overview

- type values only possible for monomorphic types
- create a type value for each monomorphic instance of a polymorphic type
- type value must be unique for a single monomorphic type
- how to know if a type value of a monomorphic instance already exists?
- using `Datatype.Polymorphic`, `Datatype.Polymorphic2` instead of `Datatype.Make` solves this issue.

Datatype

Polymorphism

Use

```
module Rational =
  Datatype.Pair(Datatype.Int)(Datatype.Int)
let rational =
  Datatype.pair Datatype.int Datatype.int

module Rational_string_map =
  Rational.Map.Make(String)

let rational_list_list2unit =
  Datatype.func
    (Datatype.list (Datatype.list rational))
    Datatype.unit
```

Dynamic Linking

Dynamic Linking

- (most) plug-ins are dynamically linked against Frama-C
- their API are statically unknown
- they are dynamically registered and accessed

Export a Value

Export a Value

- Functions manipulating command line options are automatically exported
- others values must be explicitly exported thanks to `Dynamic`

```
let run () = ...
let run =
  Dynamic.register ~plugin:"Wp" "run"
    (Datatype.func Datatype.unit Datatype.unit)
    ~journalize:false
    cmdline_run
```

Use a Dynamic API

Use a Dynamic API

```
let run_wp =
  Dynamic.get ~plugin:"Wp" "run"
    (Datatype.func Datatype.unit Datatype.unit)

let main () = ...; run_wp (); ...
```

Abstract Type


```
(* plugin.ml *)
module Rational = struct
  type rational = int * int
  include Datatype.Make_with_collections
    (struct let name = "Rational.t" ... end)
  let make n d = ...
  let make =
    Dynamic.register
      ~plugin:"Plugin" "Rational.make"
      (Datatype.func2
         Datatype.int Datatype.int ty)
      ~journalize:false
      make
end
```

Abstract Type

- Cannot directly access to `Rational.ty`

```
(* user.ml *)
module Rational =
  Type.Abstract
    (struct let name = "Rational.t" end)

let make_rational =
  Dynamic.get
    ~plugin:"Plugin" "Rational.make"
    (Datatype.func2
       Datatype.int Datatype.int Rational.ty)

let half = make_rational 1 2
```

Journalisation

Journalisation

- must provide ocaml pretty-printers
- set labeled argument `journalize`

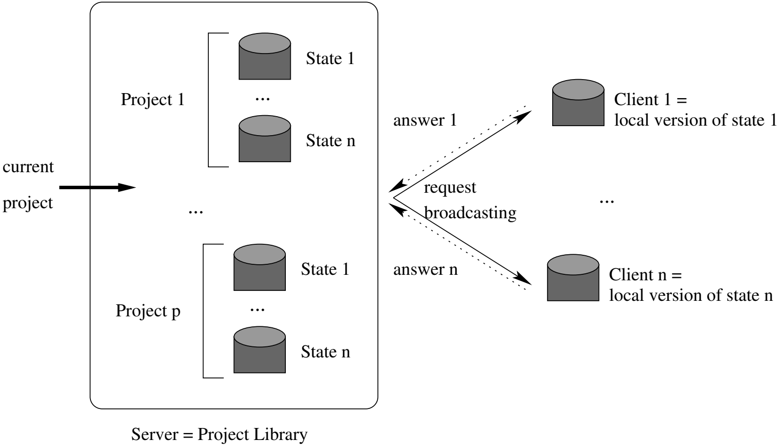
```
let run () = ...
let run =
  Dynamic.register ~plugin:"Wp" "run"
    (Datatype.func Datatype.unit Datatype.unit)
    ~journalize:true
    cmdline_run
```

Project Overview

- *Frama-C* may handle several ASTs in the same session
- a project groups together one AST with all the global data attached to it
- examples of such data are
  - the AST itself
  - kernel tables like those of kernel functions and annotations
  - command line options
  - results of analyzers
- such data are called states
- by default, each operation are applied on the current project

Overview

Client/Server View



- delayed synchronization between client *i* and server' state *i* of the current project

State

- each time you create a global data, ask yourself: "is this data part of a project or common to all projects?"
- most often, it is really part of a project
- in such cases, you have to create a *projectified state* (otherwise use standard *OCaml* datastructures like references or hashtables)

State

- module `State_builder`
- a state is a module created through functor application
- low-level functor `State_builder.Register`
- several high-level functors
  - `State_builder.Ref`
  - `State_builder.Option_ref`
  - `State_builder.Set_ref`
  - `State_builder.Hashtbl`
  - `State_builder.Queue`
  - `State_builder.Counter`
  - ...
- much simpler to use them (prefer a reference to a record than a mutable record, even if less efficient...)

State

State

Registration in Practice

```
module My_bool_ref =
  False_ref(struct
    let name = "My_plugin.My_bool_ref"
    let dependencies = []
    let kind = `Correctness
  end)
```

State

State

Registration in Practice (2)

```
type callstack =
  (Cil_types.stmt * Kernel_function.t) list

module My_callstack =
  State_builder.Ref
    (Datatype.List
      (Cil_datatype.Stmt)(Kernel_function))
    (struct
      let name = "My_plugin.My_callstack"
      let dependencies =
        [ Ast.self; Kernel_function.self ]
      let kind = `Correctness
      let default () = []
    end)
```

State

State

Registration in Practice (3)

```
module My_hashtbl =
  State_builder.Hashtbl
    (Cil_datatype.Stmt.Hashtbl)
    (Datatype.String)
    (struct
      let name = "My_plugin.My_hashtbl"
      let dependencies = [ Ast.self ]
      let kind = `Correctness
      let size = 17
     end)
```

State

State

Use

```
open Cil_types
let _ = object (self)
  inherit Visitor.frama_c_inplace
  method vinst = function
    | Call(_ret_lval,
           { enode = Lval(Var v, NoOffset) },
           _args,
           _loc) →
      My_callstack.set
        ((Extlib.the self#current_stmt,
          Globals.Functions.get v)
        :: (My_callstack.get ()));
      Cil.SkipChildren
    | _ → Cil.SkipChildren
end
```

Project Operations

Project Operations

- Project.current
- Project.create, Project.remove
- Project.copy
- Project.save, Project.load
- Project.set_current, Project.on

```
let main () =
  let p =
    !Db.Sparecode.get
      ~select_annot:false
      ~select_slice_pragma:false
  in
  Project.on p !Db.Value.compute ()
```

State Selection

Overview

- project operations may be applied only on some states
- such a set of states is called a *state selection*
- a way to improve efficiency
- a way to easily implement some operations over states (like clearing)
- must preserve *Frama-C*'s global consistency
- that is the *raison d'être* of *state dependencies* which allows to easily specify consistent selections

State Selection

State Selection

Example

```
(* clear value analysis' results
   and all its depending state
   in the current project *)
let selection =
  State_selection.Dynamic.with_dependencies
    !Db.Value.self
in
Project.clear ~selection ()
```

Marshaling

Marshaling

Syntactic analysis

Semantic analysis

Manipulating annotations

Real-world examples

Bibliography

Accessing Annotations

- do not read annotations directly stored in the AST
- global annotations: `Globals.Annotations`
- function contracts: `Kernel_function.get_spec`
- code annotations: `Annotations`
- visitor

Generating Annotations

- do not modify AST nodes in place
- copy visitor
- global annotation: `Globals.Annotations.add_generated`
- function contract: `Kernel_function.set_spec`
- code annotation: `Annotations.add`, `Annotations.add_assert`
  - require a list of states in argument
  - they are the states which the generation of the annotation depends on

```
let value_alarm = ... in
Annotations.add_assert
  kf stmt [ !Db.Value.self ] value_alarm
```

Upcoming Annotations

- *Frama-C* Oxygen provides a fully new API for annotations
- global annotations, function contracts and code annotations in a single module `Annotations`
- new consistent and uniform interface
- no more states in argument for code annotations
- but a so-called *emitter* for any new annotation

Property Statuses

Overview

- each plug-in may emit a (local) status for a property p, that is whether p is valid or invalid
  - "valid" means: for each execution trace from the beginning on the application to p, p is logically valid
  - "invalid" is the opposite of "valid"
  - plug-ins must be correct: it cannot says that p is valid if it is not (and conversely).
- the kernel automatically consolidates the result for each property according to all emitted statuses

From Annotations to Properties

Overview

- ACSL annotations may contain several properties (for instance, behaviors)
- module `Property` defines properties as a single datatype
- it also provides operations to convert an annotation to a propertie or a set of properties

From Annotations to Properties

Example

```
let _ = object
  inherit Visitor.frama_c_inplace as self
  method vcode_annot =
    let ppts =
      Property.ip_of_code_annot
        (Extlib.the self#current_kf)
        (Extlib.the self#current_stmt)
    in
    Pretty_utils.pp_list "%a"
      Property.pretty ppts;
    Cil.SkipChildren
end
```

Emitters

- emitters emit property statuses according to parameters
- if a correctness parameter changes, then valid statuses may become invalid (or conversely)
- if a tuning parameter changes, then only unknown statuses may be refined into valid or invalid.

```
let emitter =
  Emitter.create
    "my_emitter"
    ~correctness:[ Kernel.LibEntry.parameter ]
    ~tuning:[ My_tuning_option.paremeter ]
```

Emitting Statuses

```
let () =
  Property_status.emit
    emitter
    ~hyps:[]
    property
    Property.Dont_know
```

Emitting Statuses

Dependencies

- the local status True may depend on a set of hypotheses, that is other annotations which must be valid to ensure validity

```
let () =
  Property_status.emit
    emitter
    ~hyps:[ Property.ip_lemma "fermat_theorem"]
    property
    Property.True
```

Property Statuses

Reachability

- "p is invalid" means: it exists an execution trace from the beginning on the application to p such that p is logically invalid.
- require a proof of reachability and a proof of invalidity
- may be difficult
- 2 different local statuses:
  - `False_and_reachable`
  - `False_if_reachable` which automatically adds an hypothesis about reachability of p.

Bibliography

Reference Manuals

- L. Correnson, P. Cuoq, F. Kirchner, V. Prevosto, A. Puccetti, J. Signoles and B. Yakobowski. *Frama-C User Manual*. 2011.
- P. Baudin, P. Cuoq, J.-C. Filliâtre, C. Marché, B. Monate, Y. Moy and V. Prevosto. *ACSL: ANSI/ISO C Specification Language*. Version 1.5. 2011.
- J. Signoles, L. Correnson and V. Prevosto. *Frama-C Plug-in Development Guide*. 2011.

Bibliography

Articles

- P. Cuoq and J. Signoles. *Experience Report: OCaml for an Industrial-Strength Static Analysis Framework.* In ICFP 2009.
- J. Signoles. *Foncteurs impératifs et composés: la notion de projets dans Frama-C*. In Studia Informatica Universalis. 2009.
- J. Signoles. *Une bibliothèque de typage dynamique en OCaml.* In Studia Informatica Universalis. 2011.
- P. Cuoq, D. Doligez and J. Signoles. *Lightweight Typed Customizable Unmarshaling*. ML workshop 2011.
- L. Correnson and J. Signoles. *Combining Analyses for C Program Verification*. To appear in FMICS'12.